# Compact Identity Hashcode

## Overview

Instead of allocating storage to store objects' identity hash-code ('i-hash'), we propose to only allocate that storage when needed, that is, when an application calls System.identityHashCode(). This allows to reduce the size of object headers to 32 bits/4 bytes.

The idea is to do the following:

- When System.identityHashCode() is first called on an object and as long as the object is not moved in memory (by the GC), its i-hash gets computed and returned to the caller. In order to do this, we need to provide a new idempotent hash-code function based on the memory address of the object, because the current hash-code function essentially returns a new random number for each invocation.
- As soon as the GC moves the object to a new location, the i-hash needs to be preserved, so that future calls of System.identityHashCode() can return the same i-hash (hash-code stability is a crucial part of the contract between JVM and user code). In order to do this, GCs will allocate an extra word for the new copy of the object (if necessary, see below), and store the i-hash into an appended hidden field following all existing fields of the object/array. From that point onwards, calls to System.identityHashCode() returns the stored i-hash.
- Many objects have an alignment gap somewhere in the field layout, or at the end of the object. If the alignment gap is large enough (that is, at least 4 bytes), then the i-hash can be stored into that gap immediately and GCs don't have to expand the object.

## Object header bits

We still need 2 bits in the object header to indicate which state the object is in. The meaning of those 2 bits are the following:

- 00: The object has not yet been i-hashed. This is the default state.
- 01: The object has been i-hashed, but has not yet been moved from its original location in memory. The first call to System.identityHashCode() transitions the hash-bits from the 00 state to the 01 state, if there is no sufficient alignment gap, and return the computed i-hash. Subsequent calls of System.identityHashCode() will also re-compute and return the same i-hash (as long as the object stays at the same address, that is, as long as the object remains in the 01 state).
- 10: The object has been i-hashed and the i-hash has been stored in a hidden at the end of the object. The first call to System.identityHashCode() transitions the hash-bits from the 00 state to the 10 state, when there is sufficient alignment gap, and computes, stores and returns the i-hash in that hidden field. Also, GCs which observe the 01 hash-state in an object that needs to be moved to a new memory location, need to allocate an extra word in the object's new copy, and recompute the i-hash (from the object's original location) and store the i-hash into the hidden field in the extra word. Calls to System.identityHashCode() that observe the 10 state will always load and return the stored i-hash.

## Why is it ok to grow objects?

Question: Isn't there a problem when objects can grow, can we end up with a larger heap occupancy after GC than we had before? Why is this ok/how do we deal with this?

First of all, let's narrow the problem. We don't have to solve the legit OOM that happens because the heap is too small to accommodate all allocations. And I-hash-code is also an allocation (even with legacy i-hash - we just currently allocate it up-front). Also, existing code would continue to work with the same configuration, because we are saving (on average) 4 bytes per object, and require (maximum) 4 bytes per object for I-hash. It’s still a net-win.

What we do need to solve is malfunctioning/misbehaving GC because of unexpected heap/space/region growth.

Also, consider that this is mostly a problem for STW GCs (IMO) only. Concurrent GCs *already* have the problem that they can OOM during GC because of concurrent allocations. It’s not really interesting if that allocation comes from concurrent running Java threads or because of expanding I-hash. Concurrent GCs somehow have to deal with it already, and as stated above, we don’t have to solve legit OOMs. But again, GCs must not misbehave.

Let’s look at the various GC algorithms.

Full-GC/mark-compact (Serial, G1, Shenandoah) have an interesting property which leads to never-expanding overall heap usage. Objects are always ‘sliding’ towards the bottom (for G1 and Shenandoah it’s more complex, but conceptually the same). An object either does not move at all (e.g. already at the bottom), or moves towards the bottom. When an i-hashed object moves towards the bottom, it may have to be expanded, which makes it use one more word. This can lead to the situation that a subsequent object does *not* have to move at all. If that subsequent object is I-hashed, it would *not* have to be expanded. It can never happen that an object is moved towards the top. Therefore we can conclude that mark-compact GCs would never produce a heap that is larger than before GC, and therefore GC operation can not fail. (Yes, it is possible that it does not free up any space, but that would be a legit OOM and user should configure more heap to begin with.)

The story is different for 2-space compaction (Serial, Parallel young GCs): It is very well possible that scavenging from-space could end up requiring more memory in to-space, which would be a problem because of GC malfunction. The proposal to address this problem is to promote all objects that need to grow straight into old-gen. This seems like a reasonable thing to do - keys of a hash-table are more likely to be longer-lived anyway. Once they are in old-den, the mark-compact reasoning holds.

Region-based collectors (G1, Shenandoah, ZGC) don’t really have the problem: if they run OOM during GC because of expanding objects, it must be a legit OOM. They would not misbehave otherwise. G1 and Shenandoah even do a last-ditch mark-compact, for which the above reasoning holds.